Relating nominal and higher-order abstract syntax specifications

Gacek Andrew

Relating Nominal and Higher-order Abstract Syntax Specifications

Andrew Gacek *
INRIA Saclay – Île-de-France & LIX/École polytechnique
Palaiseau, France
gacek@lix.polytechnique.fr

Abstract

Nominal abstract syntax and higher-order abstract syntax provide a means for describing binding structure which is higher-level than traditional techniques. These approaches have spawned two different communities which have developed along similar lines but with subtle differences that make them difficult to relate. The nominal abstract syntax community has devices like names, freshness, name-abstractions with variable capture, and the ν-quantifier, whereas the higher-order abstract syntax community has devices like λ-binders, λ-conversion, raising, and the ∇-quantifier. This paper aims to unify these communities and provide a concrete correspondence between their different devices. In particular, we develop a semantics-preserving translation from αProlog, a nominal abstract syntax based logic programming language, to $\mathcal{G}^-$, a higher-order abstract syntax based logic programming language. We also discuss higher-order judgments, a common and powerful tool for specifications with higher-order abstract syntax, and we show how these can be incorporated into $\mathcal{G}^-$. This establishes $\mathcal{G}^-$ as a language with the power of higher-order abstract syntax, the fine-grained variable control of nominal specifications, and the desirable properties of higher-order judgments.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.1 [*Logic and Constraint Programming*]: Mathematical Logic; I.2.3 [*Deduction and Theorem Proving*]: Logic Programming

General Terms Languages, Theory

Keywords proof search, nominal logic, higher-order abstract syntax

* This work has been supported by INRIA through the "Equipes Associées" Slimmer and by the NSF Grant CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the author and do not necessarily reflect the views of the National Science Foundation.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

1. Introduction

Many approaches and languages have been proposed for encoding logical specifications of systems with binding. One popular approach is based on *nominal logic* which formalizes a notion of α-equivalence classes along with related devices [18]. This has led to the αProlog language which allows for executing specifications based on nominal logic [5]. Another popular approach is based on *higher-order abstract syntax* which uses a weak λ-calculus to represent binding in object systems [13, 17]. Formalizing specification based on higher-order abstract syntax requires a framework with devices for manipulating and forming judgments over λ-terms. The most notable examples of such frameworks are LF [12] and λProlog [16] which use higher-order techniques for representing both syntax and judgments.

The success of both the nominal and higher-order approaches has lead to questions regarding their relationship and relative merits such as naturalness and expressiveness. Higher-order abstract syntax provides a high-level treatment of binding and is often used with higher-order judgments to produce elegant specifications. These specifications benefit from a free notion of substitution inherited from the specification language and from nice properties which can be used when reasoning, *e.g.*, that substitution for free variables preserves the validity of judgments. On the other hand, nominal approaches require substitution issues to be dealt with manually, but allow object variables to be manipulated directly. This results in natural specifications when fine-grained control over object variables is required. The same naturalness is not found in similar situations when using higher-order judgments. This is not a limitation of the high-level treatment of binding provided by higher-order abstract syntax, but rather of the companion notion of higher-order judgments. In fact, in this paper we show that higher-order abstract syntax in a suitable framework is capable of at least the same naturalness and expressiveness as nominal logic specifications. We do this by developing and proving correct a direct translation from αProlog programs to definitions in $\mathcal{G}^-$, a logic with higher-order abstract syntax.

Higher-order judgments play an important role in higher-order abstract syntax specifications because of their elegance and nice properties. It is disappointing that they are abandoned in order to make this connection between nominal and higher-order abstract syntax specifications. We show, however, that higher-order judgments can be encoded in $\mathcal{G}^-$ so that their nice features are preserved. Thus $\mathcal{G}^-$ is a language in which the benefits of both nominal logic and higher-order judgments can be realized.

Let us consider an example to demonstrate the already close correspondence between specifications based on nominal and higher-order abstract syntax. The following αProlog program describes

type checking for λ-terms.

$$\forall G, X, T.[tc(G, var(X), T) :- lookup(X, T, G)]$$
$$\forall G, E_1, E_2, T'.[tc(G, app(E_1, E_2), T') :- \exists T.tc(G, E_1, arr(T, T')) \wedge tc(G, E_2, T)]$$
$$И x.\forall G, E, T, T'.[tc(G, lam(\langle x\rangle E), arr(T, T')) :- x\#G \wedge tc(bind(x, T, G), E, T')]$$

The last clause illustrates the specification of binding structure and features the nominal И-quantifier for fresh variable names, name-abstraction $\langle x\rangle E$ for denoting object binding structure, and the fresh relation $x\#G$ for enforcing a freshness side-condition. The same program can be specified in $\mathcal{G}^-$ using the following definitional clauses.

$$\forall G, X, T.[tc\ G\ (var\ X)\ T \triangleq lookup\ X\ T\ G]$$
$$\forall G, E_1, E_2, T'.[tc\ G\ (app\ E_1\ E_2)\ T' \triangleq \exists T.tc\ G\ E_1\ (arr\ T\ T') \wedge tc\ G\ E_2\ T]$$
$$\forall G, E, T, T'.[tc\ G\ (lam\ \lambda x.E\ x)\ (arr\ T\ T') \triangleq \nabla x.tc\ (bind\ x\ T\ G)\ (E\ x)\ T']$$

The last clause here features the ∇-quantifier for fresh variable names, a λ-binder for denoting object binding structure, and a function application $(E\ x)$ denoting a substitution. In addition, the variable quantification order in the last clause enforces the freshness side-condition: since x is quantified inside the scope of G, no instantiation for the latter can contain the former. The translation we present in this paper actually generates these definitional clauses given the original αProlog program. By studying this translation and proving it correct, we can pin down the exact relationship between the nominal and higher-order devices of these two specifications.

It is important to note that this paper is not an attempt to argue that one approach or another is irrelevant. Nominal techniques embed nicely in existing theorem provers [20], and higher-order techniques enable high-level specification and reasoning [9, 11]. In addition, this paper does not attempt to relate implementation issues associated with executing nominal and higher-order abstract syntax specifications, such as higher-order, nominal, or equivariant unification. Such relationships have been investigated in other works [3, 22].

The paper is organized as follows. We describe αProlog in Section 2, $\mathcal{G}^-$ in Section 3, and the translation in Section 4. We discuss the relationship with higher-order judgments in Section 5, and we conclude in Section 6.

2. αProlog

The syntax of αProlog is made up of terms, goals, and program clauses which are defined by the following grammars, respectively:

$$t, u ::= a \mid X \mid f(\vec{t}) \mid (a\ b) \cdot t \mid \langle a\rangle t$$
$$G ::= \top \mid p(\vec{t}) \mid a\#t \mid t \approx u \mid G \wedge G' \mid G \vee G' \mid \exists X.G \mid И a.G$$
$$D ::= И\vec{a}.\forall \vec{X}.[p(\vec{t}) :- G]$$

For terms, a and b denote *names* which are used to represent object language variables, X denotes a first-order variable, and f denotes a function symbol. Constants are encoded as function symbols which take no arguments. The construct $(a\ b) \cdot t$ denotes a *swapping* of the names a and b within the term t. The construct $\langle a\rangle t$ is called a *name-abstraction* and is used to represent object language bindings. We assume that all terms are well-typed according to a monomorphic typing discipline. We will avoid the details of typing, except to note that αProlog requires names to belong to distinguished *name types* which are not inhabited by any other terms.

$$(a\ b) \cdot a = b$$
$$(a\ b) \cdot b = a$$
$$(a\ b) \cdot a' = a' \quad (a \neq a' \neq b)$$
$$(a\ b) \cdot f(\vec{t}) = f(\overrightarrow{(a\ b) \cdot t})$$
$$(a\ b) \cdot \langle a'\rangle t = \langle (a\ b) \cdot a'\rangle (a\ b) \cdot t$$

$$\frac{a \neq b}{\models a\#b} \qquad \frac{\models a\#t_1 \quad \ldots \quad \models a\#t_n}{\models a\#f(\vec{t})}$$

$$\frac{}{\models a\#\langle a\rangle t} \qquad \frac{\models a\#b \quad \models a\#t}{\models a\#\langle b\rangle t}$$

$$\frac{}{\models a \approx a} \qquad \frac{\models t_1 \approx u_1 \quad \ldots \quad \models t_n \approx u_n}{\models f(\vec{t}) \approx f(\vec{u})}$$

$$\frac{\models t \approx u}{\models \langle a\rangle t \approx \langle a\rangle u} \qquad \frac{\models a\#u \quad \models t \approx (a\ b) \cdot u}{\models \langle a\rangle t \approx \langle b\rangle u}$$

Figure 1. Swapping, freshness, and equality for ground nominal terms

Goals are constructed from the usual logical connectives. The goal $a\#t$ is a *freshness* constraint and holds when the name a does not occur free relative to name-abstractions in t. The equality goal $t \approx u$ denotes a notion of α-convertibility which treats name-abstraction as a binder. The goal $И a.G$ represents a binding for the name a in the scope of G.

We assume a single form for program clauses. While αProlog admits richer forms, these are normalizable to the one presented here, possibly by inserting freshness constraints. The expression $p(\vec{t})$ is called the head of the clause. A predicate can appear in the head of multiple clauses. A clause is well-formed if it contains no free variables or free names. Note that the name a appears free in $\langle a\rangle a$ but not in $И a.G$ since the former is not a real binder while the latter is. We shall consider only well-formed program clauses from here onwards. An αProlog program is a set of program clauses.

We have presented the *name-restricted* subset of αProlog where a and b must be names in $(a\ b) \cdot t$, $\langle a\rangle t$, and $a\#t$. We will focus on this subset for most of the paper, but will eventually lift this restriction and treat full αProlog.

An αProlog expression is a term, list of terms, goal, or program clause. An expression is *ground* if it does not contain any free variables, though it may contain free names. We define the meaning of swapping, freshness, and equality for ground nominal terms as shown in Figure 1. We extend the notion of swapping to goals in the expected way with $(a\ b) \cdot И a'.G = И a'.(a\ b) \cdot G$ where $a \neq a' \neq b$ and with $(a\ b) \cdot \exists X.G = \exists X.(a\ b) \cdot G$. To make sense of this last equation, we define $(a\ b) \cdot X = X$, though we leave this out of the formal definition of swapping since we intend to focus on ground terms and goals. We define a permutation π as a composition of zero or more swappings and we write $\pi.e$ to denote the effect of applying the swappings in π to the expression e.

We assume the standard notions of binding for quantifiers and use $e[t/X]$ to denote capture-avoiding substitution of the term t for the variable X in the expression e. Similarly we write θ for a simultaneous substitution for zero or more variables and $e\theta$ for its

$$\frac{}{\Delta \Longrightarrow \top}\ \text{TRUE} \qquad \frac{\models a\#t}{\Delta \Longrightarrow a\#t}\ \text{FRESH} \qquad \frac{\models t \approx u}{\Delta \Longrightarrow t \approx u}\ \text{EQUAL}$$

$$\frac{\Delta \Longrightarrow G_1 \quad \Delta \Longrightarrow G_2}{\Delta \Longrightarrow G_1 \wedge G_2}\ \text{AND} \qquad \frac{\Delta \Longrightarrow G_i}{\Delta \Longrightarrow G_1 \vee G_2}\ \text{OR}$$

$$\frac{\Delta \Longrightarrow G[t/X]}{\Delta \Longrightarrow \exists X.G}\ \text{EXISTS} \qquad \frac{\Delta \Longrightarrow G}{\Delta \Longrightarrow \nabla a.G}\ \text{NEW}$$

$$\frac{\Delta \Longrightarrow \pi.(G\theta)}{\Delta \Longrightarrow p(\vec{t})}\ \text{BACKCHAIN}$$

Where $\nabla\vec{a}.\forall\vec{X}.[p(\vec{u}) :- G] \in \Delta$ and π is a permutation and θ is a substitution for $\vec{X}$ such that $\vec{t} \approx \pi.(\vec{u}\theta)$.

Figure 2. Proof rules for αProlog

application to the expression e. Note that name-abstractions are not really binders and thus substitution can cause name capture, *e.g.*, $(\langle a\rangle X)[a/X] = \langle a\rangle a$.

We view computation in αProlog as the search for a proof of the sequent $\Delta \Longrightarrow G$ where Δ is a set of program clauses and G is a goal. A sequent is well-formed if G is ground, and we shall consider only well-formed sequents from here onwards. Our view of αProlog purposefully ignores issues related to an actual implementation such as searching for instantiations for existentially quantified variables and related issues of unification [6, 22].

The proof rules for αProlog are shown in Figure 2. In the EXISTS rule, t may contain any names and similarly for the substitution θ in BACKCHAIN. In both rules, the substitutions cannot contain free variables, thereby ensuring that goals remain ground during proof search. In the BACKCHAIN rule we use the relation $\approx$ between lists of terms to mean that respective terms in the two list satisfy the $\approx$ relation.

As an example, let Δ be the set of program clauses for type checking given in the introduction and the assumed clauses for the *lookup* predicate. The object term $\lambda z.\lambda z.z$ can be assigned the type $\alpha \to \beta \to \beta$ for any types α and β. The corresponding derivation for this is shown in Figure 3. On the other hand, the term cannot be assigned the type $\alpha \to \beta \to \alpha$ when α is not equal to β. To do so would require the derivation to use the same name for both the first and second abstractions in the term. This is disallowed by the use of the freshness predicate.

An important characteristic of αProlog derivations is that they are *equivariant*, *i.e.*, unchanged by permutations of names. This property ensures that the particular choice of names used in a derivation is immaterial. More formally, one can inductively define a notion of applying a permutation to a derivation so that its structure and correctness are preserved. Assuming this, we will treat as equivalent those derivations which differ only by a permutation of names.

Cheney and Urban [6] introduce a Herbrand model based semantics for nominal logic which we can use to show the relative consistency and completeness of our presentation of αProlog. In particular, given a set of nominal logic formulas Γ and a nominal logic formula ϕ, they write $\Gamma \models \phi$ to indicate that any Herbrand model for all the elements of Γ is a model of ϕ. Using this notion, we can prove the following.

Theorem 1. *Let Δ be a set of program clauses and G a ground goal. Then $\Delta \models G$ holds if and only if $\Delta \Longrightarrow G$ has a proof.*

$$\frac{}{\longrightarrow \top}\ \top\mathcal{R} \qquad \frac{}{\longrightarrow t = t}\ =\mathcal{R}$$

$$\frac{\longrightarrow B_1 \quad \longrightarrow B_2}{\longrightarrow B_1 \wedge B_2}\ \wedge\mathcal{R} \qquad \frac{\longrightarrow B_i}{\longrightarrow B_1 \vee B_2}\ \vee\mathcal{R}$$

$$\frac{\longrightarrow B[t/x]}{\longrightarrow \exists x.B}\ \exists\mathcal{R} \qquad \frac{\longrightarrow B[a/x]}{\longrightarrow \nabla x.B}\ \nabla\mathcal{R},\ a \notin \mathrm{supp}(B)$$

$$\frac{\longrightarrow B\theta}{\longrightarrow p\,\vec{t}}\ \mathit{def}\mathcal{R}$$

Where $\forall\vec{x}.[(\nabla\vec{z}.p\ \vec{u}) \triangleq B] \in \mathcal{D}$ and θ is a substitution for $\vec{z}$ and $\vec{x}$ such that each $z_i\theta$ is a unique nominal constant, $\mathrm{supp}(\vec{x}\theta) \cap \{\vec{z}\theta\} = \emptyset$, and $\vec{t} = \vec{u}\theta$.

Figure 4. Proof rules for $\mathcal{G}^-$

Proof. The forwards direction uses the fact that a least Herbrand model exists for Δ. The backwards direction is by induction on the derivation of $\Delta \Longrightarrow G$. □

3. The Logic $\mathcal{G}^-$

The logic $\mathcal{G}^-$ is a first-order logic over a higher-order term language where specifications are encoded as fixed-point definitions for predicates. This is in contrast to languages like αProlog and λProlog which use Horn-like clauses to encode specifications. The reason for this departure is that $\mathcal{G}^-$ is actually a subset of a richer logic $\mathcal{G}$ which is designed for reasoning [9, 10], and within this larger setting, mechanisms like case-analysis and induction give a different meaning to Horn-like clauses than in pure specification logics. Our presentation of $\mathcal{G}^-$ differs from that of $\mathcal{G}$ in some regards, but it is still a proper subset and can be shown to be sound relative to $\mathcal{G}$.

The syntax of terms in $\mathcal{G}^-$ is as follows:

$$t, u ::= x \mid c \mid a \mid (t\,u) \mid \lambda x.t$$

Here x denotes a variable, c denotes a constant, and a denotes a *nominal constant*. The term $\lambda x.t$ denotes a binding for the variable x in the scope of t, and we assume the corresponding standard notations of free and bound variables and capture-avoiding substitution. The scope of a λ is as far to the right as possible. We will assume that all terms are in $\beta\eta$-long form and that all comparisons between terms are relative to the standard rules of λ-conversion. Application associates to the left and we write $p\ \vec{t}$ to abbreviate $p\,t_1\ \cdots\ t_n$. We restrict our attention to terms which are well-typed relative to a monomorphic typing system. We will avoid the details of typing, except to note that $\mathcal{G}^-$ allows nominal constants only at pre-designated *nominal types* which may or may not be inhabited by other terms.

Formulas in $\mathcal{G}^-$ are terms of a distinguished type o. We introduce the constant $\top$ of type o and the infix constants $\wedge$ and $\vee$ of type $o \to o \to o$. For each type τ that does not contain o we include the equality constant $=_\tau$ of type $\tau \to \tau \to o$ and the constants $\exists_\tau$ and ∇_τ of type $(\tau \to o) \to o$. We place the further restriction on ∇_τ that τ must be a nominal type. We drop subscripts when they can be inferred from the context. We abbreviate the formulas $\exists(\lambda x.t)$ and $\nabla(\lambda x.t)$ as $\exists x.t$ and $\nabla x.t$, respectively. In summary, the formulas of $\mathcal{G}^-$ are described by the following grammar.

$$B, C ::= \top \mid p\,\vec{t} \mid t = u \mid B \wedge C \mid B \vee C \mid \exists x.B \mid \nabla z.B$$

Here p denotes any additional predicate symbol, *i.e.*, constant of type $\tau_1 \to \ldots \to \tau_n \to o$.

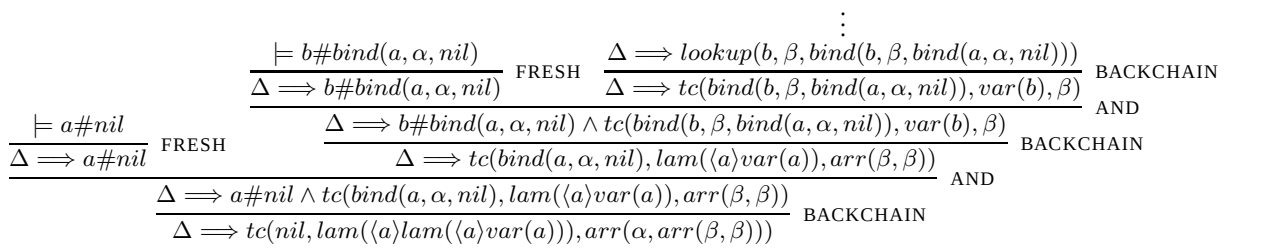

Figure 3. A derivation of tc in αProlog

We use ∇ to quantify over fresh nominal constants. The treatment of ∇ in $\mathcal{G}^-$ is based on the so-called *nominal* ∇-quantifier [19] rather than the earlier *minimal* ∇-quantifier [14]. The essential difference is that the nominal ∇-quantifier admits exchange, $\nabla x.\nabla y.B \equiv \nabla y.\nabla x.B$, and weakening and strengthening, $\nabla x.B \equiv B$ if x does not appear in B. We prefer the nominal treatment since nominal constants are often used to represent variable names, and these equivalences match our intuitions about fresh variable names. In addition, the nominal treatment often results in simplified meta-theory and reasoning.

We define the support of a term as the nominal constants which appear in it:

$$\begin{array}{ll} \mathrm{supp}(x) = \mathrm{supp}(c) = \emptyset & \mathrm{supp}(a) = \{a\} \\ \mathrm{supp}(t\ u) = \mathrm{supp}(t) \cup \mathrm{supp}(u) & \mathrm{supp}(\lambda x.t) = \mathrm{supp}(t) \end{array}$$

We define the support of a list of terms as the union of their supports.

Specifications are realized in $\mathcal{G}^-$ through *fixed-point definitions*. Fixed-point definitions are given by a set of *definitional clauses*, each of the following form:

$$\forall \vec{x}.[(\nabla \vec{z}.p\ \vec{t}) \triangleq B]$$

Here $\nabla \vec{z}.p\ \vec{t}$ and B must be formulas with empty support and free variables only among $\vec{x}$. The formula $\nabla \vec{z}.p\ \vec{t}$ is called the head of the clause. A predicate symbol p may appear in the head of multiple clauses. The logic $\mathcal{G}^-$ is parametrized by a set of definitional clauses which we will call $\mathcal{D}$.

We view computation in $\mathcal{G}^-$ as the search for a proof of the sequent $\longrightarrow B$ where B is a closed formula which may contain nominal constants. The proof rules for $\mathcal{G}^-$ are presented in Figure 4. In the $\exists\mathcal{R}$ rule we assume t is a closed term which may contain any nominal constants and similar for the substitution in the *def*$\mathcal{R}$ rule. In the *def*$\mathcal{R}$ rule we assume a notion of pairwise equality on lists of terms. We require the support of $\vec{x}\theta$ to be disjoint from $\vec{z}\theta$ to reflect the order of quantifiers in the definitional clause. In the vocabulary of Miller *et al.* [15], the rules of $\mathcal{G}^-$ allow only uniform proofs and thus $\mathcal{G}^-$ is an abstract logic programming language.

Assuming the definition of tc from the introduction and a suitable definition of $lookup$, Figure 5 shows that $\lambda z.\lambda z.z$ can be assigned the type $\alpha \to \beta \to \beta$. Note that it is not possible to assign the type $\alpha \to \beta \to \alpha$ when α is not equal to β due to side-condition on the $\nabla\mathcal{R}$ rule.

Derivations in $\mathcal{G}^-$ are equivariant with respect to nominal constants, *i.e.*, the particular nominal constants used in a $\mathcal{G}^-$ derivation are irrelevant. Given a permutation of nominal constants, one can inductively define a notion of applying that permutation to a derivation so that the structure and correctness are preserved. Thus we will treat as equivalent those derivations which differ only by a permutation of nominal constants.

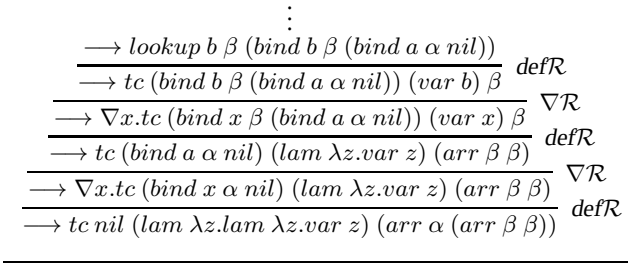

Figure 5. A derivation of tc in $\mathcal{G}^-$

4. The Translation

Looking at the rules for αProlog and $\mathcal{G}^-$ we can already see a strong similarity. In large part, this is because we have developed a view of αProlog free from implementation details and have carved out $\mathcal{G}^-$ from the richer logic of $\mathcal{G}$. We have, however, remained faithful to both languages.

One might expect a very simple translation from αProlog to $\mathcal{G}^-$ which maps M to ∇, $\approx$ to $=$, names to nominal constants, and name-abstraction to λ-abstraction. This is not far from the truth, but there is an important nuance concerning the treatment of abstractions in the two systems. In essence, the name-abstraction of αProlog allows names to be captured during substitution while the λ-abstraction requires capture-avoiding substitution. For example, consider the αProlog goal $\mathsf{M}a.\exists X.(\langle a\rangle X \approx \langle b\rangle b)$. This goal is provable using NEW and EXISTS with X as a thus yielding $\langle a\rangle a \approx \langle b\rangle b$ which is true. Now, a naive and incorrect translation of the original goal into $\mathcal{G}^-$ might produce $\nabla a.\exists X.(\lambda a.X = \lambda b.b)$. Notice that the two occurrences of a in this goal represent distinct binders and thus the goal is equivalent to $\nabla y.\exists X.(\lambda z.X = \lambda b.b)$. This formula is not provable in $\mathcal{G}^-$ since capture-avoiding substitution does not allow any value for X to be captured by the binder for z. Instead, we need a translation which makes the possible variable captures in αProlog explicit. For instance, the original goal may be translated to essentially $\exists X.(\lambda a.X\ a = \lambda b.b)$ which has the solution $X = \lambda z.z$. In this formula, the variable X has been *raised* over a to indicate its possible dependence on it. This is a standard technique which is used in relating nominal and higher-order term languages [22]. In the actual translation we will use raising to encode all such dependencies.

The translation from αProlog to $\mathcal{G}^-$ is presented in Figure 6 and makes use of some new notation which we define now.

In the translation for terms, we map names to nominal constants, and for simplicity we overload notation to use the same names for both. We use the same overloading for bound variables and function symbols. We abuse notation in our translation to allow bound variables in αProlog to be raised over nominal constants. This is just an intermediate form which is translated to a $\mathcal{G}^-$ bound variable

$$\phi(a) = a \qquad \phi(X\ \vec{a}) = X\ \vec{a} \qquad \phi((a\ b) \cdot t) = (a\ b) \cdot \phi(t) \qquad \phi(f(\vec{t})) = f\ \overrightarrow{\phi(t)} \qquad \phi(\langle a \rangle t) = \lambda a.\phi(t)$$

$$\begin{aligned}
\phi_{\vec{a}}\ (p\ \vec{t}) &= \nabla \vec{a}.p\ \overrightarrow{\phi(t)} & \phi_{\vec{a}}\ (G_1 \wedge G_2) &= \phi_{\vec{a}}(G_1) \wedge \phi_{\vec{a}}(G_2) \\
\phi_{\vec{a}}\ (\top) &= \top & \phi_{\vec{a}}\ (G_1 \vee G_2) &= \phi_{\vec{a}}(G_1) \vee \phi_{\vec{a}}(G_2) \\
\phi_{\vec{a}}\ (a\#t) &= \nabla \vec{a}.\mathit{fresh}\ \phi(a)\ \phi(t) & \phi_{\vec{a}}\ (\exists X.G) &= \exists X.\phi_{\vec{a}}(G[X\ \vec{a}/X]) \\
\phi_{\vec{a}}\ (t \approx u) &= \nabla \vec{a}.(\phi(t) = \phi(u)) & \phi_{\vec{a}}\ (И b.G) &= \phi_{\vec{a}b}(G)
\end{aligned}$$

$$\phi\left(И \vec{a}.\forall \vec{X}.[p(\vec{t}) :\!-\ G]\right) = \forall \vec{X}.[(\nabla \vec{a}.p\ \overrightarrow{\phi(t\sigma)}) \triangleq \phi_{\vec{a}}(G\sigma))] \text{ where } \sigma = \{X\ \vec{a}/X \mid X \in \vec{X}\}$$

Figure 6. Translation from αProlog to $\mathcal{G}^-$

with the same name raised over the same nominal constants. The translation for swappings produces a similar operation applied to a $\mathcal{G}^-$ term which we represent with the same notation. The meaning of a swapping applied to a $\mathcal{G}^-$ term is to replace all occurrences of one nominal constant with another and vice-versa. In contrast to αProlog, this operation can be carried out completely even for non-ground terms since all variables of the translation are raised over the existing nominal constants they may depend on. Thus, swapping can be carried out on the nominal constants over which variables are raised without having to know the eventual value of such variables. Lastly, the translation for name-abstractions maps them to λ-binders. Although nominal constants and bound variables are from separate syntactic classes in $\mathcal{G}^-$, we abuse notation here and in the future to write a binder for a nominal constant. The meaning of $\lambda a.t$ where a is a nominal constant is $\lambda x.t'$ where x is a fresh bound variable name and t' is the result of replacing all occurrences of a in t with x.

As indicated in the initial discussion, our translation needs to push ∇-binders underneath $\exists$-binders so that the dependencies can be made explicit. This is embodied in our translation for goals which is parametrized by a list of names which correspond to ∇-bound variables being pushed down to the atomic formulas. The following equivalences describe how the ∇-quantifier can be pushed down in a formula.

$$\begin{aligned}
\nabla x.\top &\equiv \top \\
\nabla x.(B \wedge C) &\equiv (\nabla x.B) \wedge (\nabla x.C) \\
\nabla x.(B \vee C) &\equiv (\nabla x.B) \vee (\nabla x.C) \\
\nabla x.\exists X.B &\equiv \exists X.\nabla x.B[X\ x/X]
\end{aligned}$$

In the translation for a freshness goal we make use of a distinguished predicate *fresh* which we assume is defined by the single definitional clause $\forall x.(\nabla z.\mathit{fresh}\ z\ x) \triangleq \top$. Thus $\longrightarrow \mathit{fresh}\ a\ t$ is provable if and only if a is a nominal constant which does not appear in t. Lastly, note that we translate the $И$-quantifier to ∇-quantifier by adding the quantified name to the list of eventually ∇-bound variables.

The translation for program clauses embodies essentially the same ideas as for translating goal formulas. The outer $И$-quantifiers are translated to ∇-quantifiers that need to be pushed underneath the universal quantifiers. This exchange induces the same raising substitution as when pushing ∇-quantifiers underneath existential quantifiers. When Δ is a set of program clauses we define $\phi(\Delta) = \{\phi(D) \mid D \in \Delta\}$.

Note that aside from changing the scope of ∇-quantifiers, the translation essentially preserves term and logic structure. Moreover, the translation makes the expected connections between $И$ and ∇, $\approx$ and $=$, names and nominal constants, and name-abstraction and λ-abstraction.

4.1 Examples

We now present a few examples to illustrate the translation and also to suggest some simple ways in which the results may be improved. In presenting αProlog program clauses we will elide outermost $И$ and $\forall$-quantifiers and instead use the convention that all free lowercase symbols denote names and all free uppercase symbols denote variables, all of which are captured by program clause quantifiers. For $\mathcal{G}^-$ definitions we elide the outermost $\forall$-quantifiers and assume that all capitalized symbols denote such universally quantified variables. These examples are taken from Cheney and Urban [6].

4.1.1 Type checking

Consider again the example from the introduction, where the following program clauses specify type checking for λ-terms.

$$\begin{aligned}
&tc(G, var(X), T) :\!-\ lookup(X, T, G) \\
&tc(G, app(E_1, E_2), T') :\!- \\
&\qquad \exists T.tc(G, E_1, arr(T, T')) \wedge tc(G, E_2, T) \\
&tc(G, lam(\langle x \rangle E), arr(T, T')) :\!- \\
&\qquad x\#G \wedge tc(bind(x, T, G), E, T')
\end{aligned}$$

Here we assume *lookup* is defined in the expected way. These program clauses translate to the following definitional clauses.

$$\begin{aligned}
&tc\ G\ (var\ X)\ T \triangleq lookup\ X\ T\ G \\
&tc\ G\ (app\ E_1\ E_2)\ T' \triangleq \\
&\qquad \exists T.tc\ G\ E_1\ (arr\ T\ T') \wedge tc\ G\ E_2\ T \\
&(\nabla x.tc\ (G\ x)\ (lam\ \lambda x.E\ x)\ (arr\ (T\ x)\ (T'\ x))) \triangleq \\
&\qquad (\nabla x.\mathit{fresh}\ x\ (G\ x)) \wedge \\
&\qquad (\nabla x.tc\ (bind\ x\ (T\ x)\ (G\ x))\ (E\ x)\ (T'\ x))
\end{aligned}$$

In general, a few simplifications can improve the results of the translation. We illustrate these here as applied to the last clause for tc. First, by examining types we can recognize that the object types T and T' cannot actually depend on the object term variable x and thus they do not need to be raised over it. More formally, we can use a notion like *subordination* to detect such vacuous dependencies [23]. Second, the freshness constraint on G can be solved statically: we know that G must not depend on its first argument. Finally, after performing the previous two simplifications we can recognize that the ∇-quantifier in the head of the definition is vacuous and can thus be dropped. In the end we are left with the following

definitional clause:

$$tc\ G\ (lam\ \lambda x.E\ x)\ (arr\ T\ T') \triangleq \nabla x.tc\ (bind\ x\ T\ G)\ (E\ x)\ T'$$

The definitional clauses for tc now match what one would expect to write in $\mathcal{G}^-$. Indeed, they are exactly specification presented in the introduction.

4.1.2 Polymorphic type generalization

The following program clauses describe a relationship among a polymorphic type, a list of distinct names for the binders in that type, and the resulting monomorphic type which comes from substituting the names for the binders.

$$spec(monoTy(T), nil, T) :- \top$$
$$spec(polyTy(\langle a\rangle P), cons(a, L), T) :- a\#L \wedge spec(P, L, T)$$

This is translated to the following definitional clauses:

$$spec\ (monoTy\ T)\ nil\ T \triangleq \top$$
$$(\nabla a.spec\ (polyTy\ \lambda a.P\ a)\ (cons\ a\ (L\ a))\ (T\ a)) \triangleq (\nabla a.\mathit{fresh}\ a\ (L\ a)) \wedge (\nabla a.spec\ (P\ a)\ (L\ a)\ (T\ a))$$

In this last clause we can again simplify the freshness condition to produce the following.

$$(\nabla a.spec\ (polyTy\ \lambda a.P\ a)\ (cons\ a\ L)\ (T\ a)) \triangleq \nabla a.spec\ (P\ a)\ L\ (T\ a)$$

4.1.3 Capture-avoiding substitution

The following program clauses realize capture avoiding substitution for λ-terms via a predicate $subst(E, T, X, E')$ which holds exactly when $E[T/X] = E'$.

$$subst(var(X), E, X, E) :- \top$$
$$subst(var(x), E, y, var(x)) :- \top$$
$$subst(app(M, N), E, X, app(M', N')) :- subst(M, E, X, M') \wedge subst(N, E, X, N')$$
$$subst(lam(\langle y\rangle R), E, X, lam(\langle y\rangle R')) :- y\#X \wedge y\#E \wedge subst(R, E, X, R')$$

These program clauses translate to the following definitional clauses:

$$subst\ (var\ X)\ E\ X\ E \triangleq \top$$
$$(\nabla x, y.subst\ (var\ y)\ (E\ x\ y)\ x\ (var\ y)) \triangleq \top$$
$$subst\ (app\ M\ N)\ E\ X\ (app\ M'\ N') \triangleq subst\ M\ E\ X\ M' \wedge subst\ N\ E\ X\ N'$$
$$(\nabla y.subst\ (lam\ \lambda y.R\ y)\ (E\ y)\ (X\ y)\ (lam\ \lambda y.R'\ y)) \triangleq (\nabla y.\mathit{fresh}\ y\ (X\ y)) \wedge (\nabla y.\mathit{fresh}\ y\ (E\ y)) \wedge (\nabla y.subst\ (R\ y)\ (E\ y)\ (X\ y)\ (R'\ y))$$

Simplifying the freshness constraints and removing vacuous ∇-binders in the last clause produces the following.

$$subst\ (lam\ \lambda y.R\ y)\ E\ X\ (lam\ \lambda y.R'\ y) \triangleq \nabla y.subst\ (R\ y)\ E\ X\ (R'\ y)$$

4.2 Correctness

The soundness and completeness of our translation are shown by the following results. We elide most details, but show the important lemmas and interesting cases.

Lemma 2. *Let a be a name and t a ground αProlog term. Then $\models a\#t$ holds if and only if $\longrightarrow$ fresh $\phi(a)\ \phi(t)$ has a proof in $\mathcal{G}^-$.*

Proof. Induction on t. □

Lemma 3. *Let t and u be ground αProlog terms. Then $\models t \approx u$ holds if and only if $\phi(t) = \phi(u)$.*

Proof. Induction on t. □

We define the support of an αProlog term as the set of all names which appear free relative to name-abstractions, *i.e.*, $\mathrm{supp}(t) = \{a \mid a\#t \text{ does not hold}\}$. This is consistent with the definition of support for $\mathcal{G}^-$ terms since $\phi(\mathrm{supp}(t)) = \mathrm{supp}(\phi(t))$. For a substitution θ we define $\phi(\theta) = \{\phi(t)/\phi(x) \mid t/x \in \theta\}$ and $\mathrm{supp}(\theta) = \bigcup_{t/x\in\theta} \mathrm{supp}(t)$.

Lemma 4. *Let t be an αProlog term and θ a substitution, then $\phi(t\theta) = \phi(t)\phi(\theta)$.*

Proof. Induction on t. □

Lemma 5. *Let G be an αProlog goal, θ a substitution, and $\vec{a}$ a list of names such that $\mathrm{supp}(\theta) \cap \{\vec{a}\} = \emptyset$, then $\phi_{\vec{a}}(G\theta) = \phi_{\vec{a}}(G)\phi(\theta)$.*

Proof. Induction on G. Consider when $G = b\#t$. Then $\phi_{\vec{a}}(G\theta) = \nabla\vec{a}.\mathit{fresh}\ \phi(b\theta)\ \phi(t\theta) = \nabla\vec{a}.(\mathit{fresh}\ \phi(b)\ \phi(t))\phi(\theta)$. Since $\mathrm{supp}(\theta)\cap\{\vec{a}\} = \emptyset$, we can move the substitution outside of the ∇-binder to obtain $(\nabla\vec{a}.(\mathit{fresh}\ \phi(b)\ \phi(t)))\phi(\theta) = \phi_{\vec{a}}(G)\phi(\theta)$. □

Theorem 6. *Let Δ be a set of program clauses, G a ground goal, and $\vec{a}$ a list of distinct names. There is a proof of $\Delta \Longrightarrow G$ if and only if there is a proof of $\longrightarrow \phi_{\vec{a}}(G)$ assuming the definitional clauses $\phi(\Delta)$ and the clause for fresh.*

Proof. In the forwards direction, the proof is by induction on the height of the αProlog proof. First consider when the proof ends with FRESH so that $G = b\#t$. Then it must be that $\models b\#t$ which means $\longrightarrow$ *fresh* $\phi(b)\ \phi(t)$ has a proof. Thus $\longrightarrow \phi_{\vec{a}}(G)$ which is $\longrightarrow \nabla\vec{a}.\mathit{fresh}\ \phi(b)\ \phi(t)$ also has a proof. The cases for TRUE, EQUAL, AND, and OR are similarly easy.

Suppose the proof ends with EXISTS so that $G = \exists X.G'$ and $\Delta \Longrightarrow G'[t/X]$ has a proof for some t. By induction we know $\longrightarrow \phi_{\vec{a}}(G'[t/X])$ has a proof. We would like to move the substitution outside of ϕ, but we cannot do so unless the support of the substitution is disjoint from $\vec{a}$. Abusing notation, we do this by splitting the substitution into two parts. That is, we know $\longrightarrow \phi_{\vec{a}}(G'[X\ \vec{a}/X][\lambda\vec{a}.t/X])$ has a proof. Now the $\vec{a}$ are not free in the second substitution and thus we can apply Lemma 5 to conclude that $\longrightarrow \phi_{\vec{a}}(G'[X\ \vec{a}/X])[\lambda\vec{a}.\phi(t)/\phi(X)]$ has a proof. Thus $\longrightarrow \exists X.\phi_{\vec{a}}(G'[X\ \vec{a}/X])$ has a proof and this is the same as $\longrightarrow \phi_{\vec{a}}(\exists X.G')$. The case for BACKCHAIN is similar in spirit, but more complex in the details.

Lastly, suppose the proof ends with NEW so that $G = \text{И}b.G'$ and $\Delta \Longrightarrow G'$ has a proof. By induction $\longrightarrow \phi_{\vec{a}b}(G')$ also has a proof which is the same as $\longrightarrow \phi_{\vec{a}}(\text{И}b.G')$.

In the backwards direction, the proof is by induction on the height of the $\mathcal{G}^-$ proof with a nested induction on the size of G. Consider first when G is $t \approx u$. We assume $\longrightarrow \phi_{\vec{a}}(t \approx u)$ has a proof which means that $\longrightarrow \nabla\vec{a}.\phi(t) = \phi(u)$ has a proof and thus $\longrightarrow \phi(t) = \phi(u)$ also does. Therefore $\models t \approx u$ is true and $\Delta \Longrightarrow t \approx u$ has a proof. The cases for when G is $\top$, a freshness relation, a conjunction, or a disjunction are similarly easy.

Suppose $G = \exists X.G'$ so that $\longrightarrow \phi_{\vec{a}}(G'[X\ \vec{a}/X])[t/X]$ has a proof for some t. It must be that $t = \lambda\vec{a}.\phi(u)$ for some u. Since the support of the outer substitution is disjoint from $\vec{a}$ we can move it

inside to know $\longrightarrow \phi_{\vec{a}}(G'[X\ \vec{a}/X][\lambda\vec{a}.u/X])$ has the same proof and this is just $\longrightarrow \phi_{\vec{a}}(G'[u/X])$. By induction $\Delta \Longrightarrow G'[u/X]$ has a proof and thus $\Delta \Longrightarrow \exists X.G'$ has a proof. Again, the case for when G is a predicate is similar in spirit, but more complex in the details.

Finally suppose $G = \nabla b.G'$ so that $\longrightarrow \phi_{\vec{a}b}(G')$ has a proof. By the inner induction hypothesis, $\Delta \Longrightarrow G'$ has a proof and thus so does $\Delta \Longrightarrow \nabla b.G'$. □

4.3 Extending the translation

We now drop the name-restriction on αProlog and allow a and b to be arbitrary terms in expressions of the form $a\#t$, $(a\ b)\cdot t$, and $\langle a\rangle t$. The translation is easily extended to this richer language. Goals of the form $u\#t$ translate to *fresh* $u\ t$ as before. Terms which do not satisfy the name-restriction are first simplified so that all non-name-restricted swappings and name-abstractions appear at the top level of an equality goal. Then these translate to distinguished predicates which implement swapping and name-abstraction. This simplification is only needed statically since instantiations during proof search can only contain ground terms, thus ensuring that non-name-restricted terms do not appear dynamically.

To simplify a non-name-restricted term of the form $(u_1\ u_2)\cdot t$ or $\langle u\rangle t$ we replace it with a fresh variable, say t', conjoin the distinguished goal $t' \doteq (u_1\ u_2)\cdot t$ or $t' \doteq \langle u\rangle t$ respectively, and quantify t' appropriately (existentially when replacing terms in the body, universally when replacing terms in the head of a program clause). This forces all terms to be name-restricted except the top level of terms occurring on the right side of the $\doteq$ relation. We extend the translation to deal with this relation as follows:

$$\phi_{\vec{a}}(t' \doteq (u_1\ u_2)\cdot t) = \nabla\vec{a}.\mathit{swap}\ u_1\ u_2\ t\ t'$$
$$\phi_{\vec{a}}(t' \doteq \langle u\rangle t) = \nabla\vec{a}.\mathit{abst}\ u\ t\ t'$$

Where *swap* and *abst* are defined by the following:

$$\forall E.[(\nabla x, y.\mathit{swap}\ x\ y\ (E\,x\,y)\ (E\,y\,x)) \triangleq \top]$$
$$\forall E.[(\nabla x.\mathit{swap}\ x\ x\ (E\,x)\ (E\,x)) \triangleq \top]$$
$$\forall E.[(\nabla x.\mathit{abst}\ x\ (E\,x)\ (\lambda x.E\,x) \triangleq \top]$$

In practice it seems that non-name-restricted swappings and name-abstractions are exceedingly rare. Still, it is reassuring that such detailed manipulations of variables and binding are so succinctly described in $\mathcal{G}^-$.

The following lemmas show that our manipulations of formulas are sound and that *swap* and *abst* correctly capture swapping and name-abstraction, and therefore the extended translation can be shown to be sound and complete.

Lemma 7. *If $t \approx u$ and $\Delta[t/x] \Longrightarrow G[t/x]$ has a proof then so does $\Delta[u/x] \Longrightarrow G[u/x]$.*

Proof. Induction on the height of the proof. □

Lemma 8. *Let t' and $(a\ b)\cdot t$ be ground nominal terms. Then $\models t' \approx (a\ b)\cdot t$ holds if and only if $\longrightarrow$ swap $a\ b\ t\ t'$ is provable in $\mathcal{G}^-$.*

Proof. Induction on t. □

Lemma 9. *Let t' and $\langle a\rangle t$ be ground nominal terms. Then $\models t' \approx \langle a\rangle t$ holds if and only if $\longrightarrow$ abst $a\ t\ t'$ is provable in $\mathcal{G}^-$.*

Proof. Induction on t. □

5. Higher-order Judgments

Higher-order judgments are a common and powerful tool for specifications using higher-order abstract syntax. Among other things, they have nice properties which are often used when reasoning about such specifications. These properties do not directly hold for $\mathcal{G}^-$ specifications, so one may question if we have to give them up in any system which admits translations from nominal logic specifications. This is not the case: higher-order judgments can be encoded in $\mathcal{G}^-$ while keeping their nice properties. Although this does not provide a direct connection between nominal logic specifications and higher-order judgments, it does show how the expressiveness and benefits of both approaches can coexist.

Our translation produces specifications using *weak higher-order abstract syntax*, *i.e.*, using abstractions only at distinguished variable types. With higher-order judgments it is much more common to use *full higher-order abstract syntax*, *i.e.*, where abstraction is used at the same type as the terms being constructed. For example, λ-terms would be represented using only the following two constants.

$$\mathit{app} : tm \to tm \to tm \qquad \mathit{lam} : (tm \to tm) \to tm$$

This representation provides a free notion of capture-avoiding substitution based on meta-level β-reduction whereas a weak higher-order abstract syntax encoding would provide only variable for variable substitution. The following example defines evaluation for λ-terms using the free notion of capture-avoiding substitution.

$$\mathit{eval}\ (\mathit{lam}\ \lambda x.R\,x)\ (\mathit{lam}\ \lambda x.R\,x) \triangleq \top$$
$$\mathit{eval}\ (\mathit{app}\ M\ N)\ V \triangleq$$
$$\exists R.\mathit{eval}\ M\ (\mathit{lam}\ \lambda x.R\,x) \wedge \mathit{eval}\ (R\,N)\ V$$

With full higher-order abstract syntax, we can still distinguish variables from other terms by using a definition such as $(\nabla x.\mathit{name}\ x) \triangleq \top$ which holds only on nominal constants. For example, the specification of type checking for λ-terms represented using full higher-order abstract syntax can be written as follows:

$$\mathit{tc}\ G\ X\ T \triangleq \mathit{name}\ X \wedge \mathit{lookup}\ X\ T\ G$$
$$\mathit{tc}\ G\ (\mathit{app}\ E_1\ E_2)\ T' \triangleq$$
$$\exists T.\mathit{tc}\ G\ E_1\ (\mathit{arr}\ T\ T') \wedge \mathit{tc}\ G\ E_2\ T$$
$$\mathit{tc}\ G\ (\mathit{lam}\ \lambda x.E\,x)\ (\mathit{arr}\ T\ T') \triangleq$$
$$\nabla x.\mathit{tc}\ (\mathit{bind}\ x\ T\ G)\ (E\,x)\ T'$$

Thus in $\mathcal{G}^-$ we have the same naturalness and expressiveness with full higher-order abstract syntax as with the weaker version.

Moving to higher-order judgments, the specification for type checking λ-terms can be written as follows in λProlog [16]:

$$\mathit{tc}\ (\mathit{app}\ M\ N)\ B :- (\mathit{tc}\ M\ (\mathit{arr}\ A\ B) \wedge \mathit{tc}\ N\ A)$$
$$\mathit{tc}\ (\mathit{lam}\ \lambda x.R\,x)\ (\mathit{arr}\ A\ B) :- (\forall x.\mathit{tc}\ x\ A \Rightarrow \mathit{tc}\ (R\,x)\ B)$$

Here tc does not carry around an explicit typing context. Instead, the context in λProlog is used to remember typing assignments for bound variables: the $\forall$-quantifier encodes fresh variable names and the $\Rightarrow$ connective encodes hypothetical assumptions. Besides elegance, the real benefit of this encoding is that the underlying logic enjoys meta-theoretic properties such as cut-admissibility and the preservation of provability under instantiations for universal variables. As a result, if one wants to reason about this specification they obtain a free object-level substitution result for tc which says, roughly, if $tc\ (\mathit{lam}\ \lambda x.R\,x)\ (\mathit{arr}\ A\ B)$ and $tc\ N\ A$ are both derivable, then $tc\ (R\,N)\ B$ is also derivable. Such substitution lemmas are quite common and useful, for example, in showing that evaluation preserves typing.

Higher-order judgments do not allow one to directly distinguish between free variables, and this can result in awkward specifications in some instances. For example, the following αProlog clauses specify a notion of inequality over λ-terms:

$$aneq\ (var\ a)\ (var\ b) :- \top$$
$$aneq\ (app\ M_1\ N_1)\ (app\ M_2\ N_2) :- aneq\ M_1\ M_2$$
$$aneq\ (app\ M_1\ N_1)\ (app\ M_2\ N_2) :- aneq\ N_1\ N_2$$
$$aneq\ (lam\ \langle a \rangle R_1)\ (lam\ \langle a \rangle R_2) :- aneq\ R_1\ R_2$$
$$aneq\ (var\ X)\ (app\ M\ N) :- \top$$
$$\vdots$$

There is no equally natural way to express this with higher-order judgments due to the need to distinguish between variables in the first clause (see [6] for an example encoding in λProlog). In fact, the ability to distinguish between variables is fundamentally at odds with the idea of a free substitution property for variables since such substitutions may cause two different variables to be instantiated to the same term. Thus, higher-order judgments necessarily lack a degree of naturalness for some specifications.

It is important to note that in a dependently-typed system like LF higher-order abstract syntax and higher-order judgments collapse into a single notion [12]. While this provides for a very elegant system, we should not let it confuse us between these two notions. As our translation has shown, higher-order abstract syntax allows for specifications that are at least as natural and expressive as what is possible in αProlog, while higher-order judgments may occasionally fall short.

We now propose a specification methodology which allows one to use the full natural expressiveness of definitions in $\mathcal{G}^-$ while still being able to take advantage of higher-order judgments so that we can benefit from their elegance and associated properties when reasoning. The idea is to encode an interpreter for higher-order judgments as a definition in $\mathcal{G}^-$ and use this to encode particular higher-order judgment specifications. In the full logic $\mathcal{G}$ which is used for reasoning, one can prove general instantiation and cut-admissibility properties for the encoding of higher-order judgments. These properties are then inherited for free by any specification written using higher-order judgments.

For simplicity of presentation we consider only the second-order fragment of λProlog. It is possible to encode full higher-order λProlog, but second-order is sufficient for the majority of examples. The encoding of second-order λProlog into $\mathcal{G}^-$ is presented in Figure 7. In this encoding :: is an infix constructor for lists, and $\langle \cdot \rangle$ is used to distinguish atomic formulas. Since we are considering only second-order λProlog, we assume A is atomic in $A \Rightarrow B$. The formula *seq* L G will hold when the λProlog formula G is provable from the atomic assumptions in L and the clauses of our particular specification. These latter clauses are encoded via the predicate *prog* which holds on the head and body of each encoded clause. For example, the clauses for the tc predicate are encoded into the following *prog* clauses:

$$prog\ (tc\ (app\ M\ N)\ B)$$
$$(\langle tc\ M\ (arr\ A\ B) \rangle \wedge \langle tc\ N\ A \rangle) \triangleq \top$$

$$prog\ (tc\ (lam\ \lambda x.R\,x)\ (arr\ A\ B))$$
$$(\forall x.tc\ x\ A \Rightarrow \langle tc\ (R\,x)\ B \rangle) \triangleq \top$$

The *seq* encoding of second-order λProlog retains the desirable properties of the logic which we formally state below. Moreover, these properties can be proven completely within the full logic $\mathcal{G}$.

Lemma 10 (Instantiation)**.** *Let c be a nominal constant and t a term of the same type. If $\longrightarrow$ seq L G then $\longrightarrow$ seq $L[t/c]$ $G[t/c]$.*

$$member\ B\ (B :: L) \triangleq \top$$
$$member\ B\ (C :: L) \triangleq member\ B\ L$$

$$seq\ L\ \top \triangleq \top$$
$$seq\ L\ (B \wedge C) \triangleq seq\ L\ B \wedge seq\ L\ C$$
$$seq\ L\ (B \vee C) \triangleq seq\ L\ B \vee seq\ L\ C$$
$$seq\ L\ (A \Rightarrow B) \triangleq seq\ (A :: L)\ B$$
$$seq\ L\ (\forall x.Bx) \triangleq \nabla x.seq\ L\ (B\,x)$$
$$seq\ L\ \langle A \rangle \triangleq member\ A\ L$$
$$seq\ L\ \langle A \rangle \triangleq \exists B.prog\ A\ B \wedge seq\ L\ B$$

Figure 7. Second-order λProlog in $\mathcal{G}^-$

Lemma 11 (Cut admissibility)**.** *If $\longrightarrow$ seq $(A :: L)$ G and $\longrightarrow$ seq L $\langle A \rangle$ then $\longrightarrow$ seq L G.*

Lemma 12 (Monotonicity)**.** *If $\longrightarrow$ seq L G and every element of L appears in K then then $\longrightarrow$ seq K G.*

When working with full higher-order abstract syntax, Lemmas 10 and 11 are quite powerful and provide the object-level substitution lemmas described earlier. Although a definition like *seq* could be encoded in αProlog or a similar language, the corresponding lemmas would not be as useful because of the weaker notion of substitution.

A fundamental restriction of the proposed method for specification is that while $\mathcal{G}^-$ definitions can make use of higher-order judgments, it is not possible for higher-order judgments to use $\mathcal{G}^-$ definitions. This restriction is inherent in our encoding via *seq* and is necessary to preserve the desirable meta-properties of higher-order judgments. Note, however, that this is a restriction on "control-flow" and not "data-flow" since we may still use higher-order judgments to specify a value which is then fed into a $\mathcal{G}^-$ specification.

6. Conclusions and Future Work

Our translation provides a direct and concrete connection between αProlog and $\mathcal{G}^-$ including a tight mapping from the devices of the former to the corresponding devices of the latter. In particular, our translation provides an understanding of the relationship between the $\mathit{И}$ and ∇-quantifiers. When used at distinguished variable types, the ∇-quantifier exactly captures the meaning of the $\mathit{И}$-quantifier, at least from the perspective of specification. However, the ∇-quantifier can also be used at types which contain other constructors which is essential for full higher-order abstract syntax and which is not possible with current understandings of the $\mathit{И}$-quantifier. In addition, using raising and the *fresh* predicate, the ∇-quantifier can be freely moved up and down in a formula whereas the $\mathit{И}$-quantifier is always given a large scope since nominal logic does not have raising and therefore cannot push the $\mathit{И}$-quantifier underneath other quantifiers.

Through our translation we have also shown that higher-order abstract syntax specifications can have at least the same naturalness and expressiveness as nominal logic specifications. The resulting specifications are based on weak higher-order abstract syntax, but we have argued that the same qualities can be found with full higher-order abstract syntax. Therefore, despite being a very high-level approach to binding, higher-order abstract syntax can still be used naturally in specifications which demand a fine-grained control over variables.

We have acknowledged the occasional failings of higher-order judgments to naturally capture some aspects of specifications involving binding. We have proposed a method which allows one to use higher-order judgments when relevant and a stronger specification language when preferred. This method allows one to benefit from the elegance of higher-order judgment during specifications and from their associated meta-properties during reasoning.

Thus we have presented the logic programming language $\mathcal{G}^-$ which has the power of higher-order abstract syntax, the fine-grained variable control of nominal specifications, and the ability to capture the desirable properties of higher-order judgments.

It seems possible to develop a fairly direct reverse translation from $\mathcal{G}^-$ to αProlog in the cases where only weak higher-order abstract syntax is used. We have not pursued this line of work since full higher-order abstract syntax is more common and is required to reap the complete benefits of using higher-order judgments. Towards this, Gabbay and Cheney have developed a translation from $FO\lambda^{\nabla}$, a first-order logic with λ-terms, full higher-order abstract syntax, and the ∇-quantifier, to a variant of nominal logic with λ-terms and the $\mathit{И}$-quantifier [2, 7]. Their translation makes a similar connection between the $\mathit{И}$- and ∇-quantifiers as in this paper though in the opposite direction. However, the presence of λ-terms in their nominal logic is unorthodox and it would be interesting to see a similar result for a more traditional nominal logic.

We have ignored issues of executing specifications in our discussions, but we consider them briefly now. Most αProlog specifications can be efficiently executed [6, 22], while some require an expensive operation known as *equivariant unification* to backchain on clauses with $\mathit{И}$-quantifiers in the head [4]. It should also be possible to efficiently execute $\mathcal{G}^-$ definitions in a similar way, with similar issues when unfolding definitional clauses with ∇-quantifiers in the head. The difficulty of this corresponds roughly with that of solving equivariant unification problems in αProlog. However, specifications in $\mathcal{G}^-$ tend to use ∇-quantifiers in the head of definitions less often than αProlog uses $\mathit{И}$-quantifiers in the head of clauses since $\mathcal{G}^-$ has real λ-binders whereas αProlog must use $\mathit{И}$-quantification and name-abstraction to represent a binder. To efficiently execute such benign uses of the $\mathit{И}$-quantifier in the head of clauses, researchers have studied the notion of $\mathit{И}$-*goal formulas* [6, 21] which are essentially those which translate to $\mathcal{G}^-$ definitions without ∇-quantifiers in the head of clauses except for the distinguished *fresh*, *swap*, and *abst* predicates.

In the worst case, our translation may produce a quadratic increase in the size of formulas due to raising. In practice, this does not appear be an issue for several reasons. First, most specifications mention few object variables per formula and thus the amount of raising required is fairly limited. Second, by making dependencies explicit via raising we are able to statically solve freshness constraints as shown in the examples. This removes some raising and decreases the number of atomic formulas in a definition. Third, we are not proposing that specifications in $\mathcal{G}^-$ be written as if translated from αProlog. Specifications that instead use full higher-order abstract syntax and the corresponding notion of substitution inherit free implementation benefits. For example, an implementation can lazily apply substitutions which may result in significant performance improvements. Further research is needed to accurately assess the relative efficiency of nominal and higher-order abstract syntax specifications.

The Abella system [8] is a theorem prover for the logic $\mathcal{G}$ and supports the proposed method of specification where higher-order judgments are mixed with $\mathcal{G}$ definitions. In practice, we have found that this hybrid style provides a nice compromise between elegance and practicality. The reasoning over such specifications benefits from the meta-properties of higher-order judgments as expected and from the naturalness of specifications which directly use the features of $\mathcal{G}$. Abella can also execute these specifications, though it is not optimized for this. Through the translation given in this paper, it is possible to use Abella to execute and reason about αProlog specifications. An interesting direction for future work would be to assess such capabilities and to possibly develop them explicitly within Abella.

Finally, we note that higher-order judgments are incorporated in $\mathcal{G}$ via a definition and not by using the universal quantifier and implication of $\mathcal{G}$. Indeed, the latter devices have a much different behavior in $\mathcal{G}$ than in higher-order judgments. In higher-order judgments, universal quantification denotes a generic quantification and implication denotes a fixed assumption. In $\mathcal{G}$, universal quantification denotes a quantification over each and every possible value and implication restricts attention to worlds in which the hypotheses are provable. The Bedwyr system is based on a a $\mathcal{G}$-like logic and uses these devices to encode and execute specifications of model-checking behavior such as bisimulation for the finite π-calculus [1].

References

[1] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conference on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397. Springer, 2007.

[2] J. Cheney. A simpler proof theory for nominal logic. In *8th International Conference on the Foundations of Software Science and Computational Structures (FOSSACS)*, volume 3441 of *LNCS*, pages 379–394. Springer, 2005.

[3] J. Cheney. Relating higher-order pattern unification and nominal unification. In L. Vigneron, editor, *Proceedings of the 19th International Workshop on Unification, UNIF'05*, pages 104–119, 2005.

[4] J. Cheney. Equivariant unification. *Journal of Automated Reasoning*, December 2009. (published online).

[5] J. Cheney and C. Urban. Alpha-prolog: A logic programming language with names, binding, and alpha-equivalence. In B. Demoen and V. Lifschitz, editors, *Logic Programming, 20th International Conference*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004.

[6] J. Cheney and C. Urban. Nominal logic programming. *ACM Trans. Program. Lang. Syst.*, 30(5):1–47, 2008. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1387673.1387675.

[7] M. J. Gabbay and J. Cheney. A sequent calculus for nominal logic. In *Proc. 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 139–148, 2004.

[8] A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

[9] A. Gacek, D. Miller, and G. Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in Electronic Notes in Theoretical Computer Science, pages 85–100, 2008.

[10] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.

[11] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[13] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, Sept. 1987.

[14] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.

[15] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[16] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Aug. 1988. MIT Press.

[17] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[18] A. M. Pitts. Nominal logic, A first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[19] A. Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, 2006.

[20] C. Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[21] C. Urban and J. Cheney. Avoiding equivariance in alpha-prolog. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2005. ISBN 3-540-25593-1.

[22] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

[23] R. Virga. *Higher-order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.